Interactive systems with registers and voices

Gheorghe Stefanescu*
School of Computing
National University of Singapore

Abstract. We present a model and a core programming language appropriate for modeling and programming interactive computing systems.

The model consists of rv-systems (interactive systems with registers and voices); it includes register machines, is space-time invariant, is compositional, may describe computations extending in both time and space, and is applicable to open, interactive systems. To achieve modularity in space the model uses voices (a voice is the time dual of a register) - they provide a high level organization of temporal data and are used to describe interaction interfaces of processes.

The programming language uses novel techniques for syntax and semantics to support computation in space paradigm. We describe rv-programs and base their syntax and operational semantics on FIS-es (finite interactive systems) and their grid languages (a FIS is a kind of 2-dimensional automaton specifying both control and interaction used in rv-programs).

We also present specification techniques for rv-systems, using relations between input registers and voices and their output counterparts. The paper includes simple specifications for an OO-system and for an interactive game.

Keywords: interactive systems, programming languages, syntax, semantics, temporal specifications, registers, voices, object-oriented systems, finite interactive systems, rv-systems, rv-programs, grids, space-time duality

1. Introduction

Interactive systems are omnipresent - they range from describing low level interacting processes on the same machine, cluster, or distributed system to describing communicating agents in the Internet, human-computer, or human-human interaction.

*This research has been partially conducted while the author was visiting professor at School of Computing, National University of Singapore. Permanent Address: Prof. Dr. G. Stefanescu, Faculty of Mathematics and Computer Science, University of Bucharest, Str. Academiei 14, RO-70101, Bucharest, Romania; Email: gheorghe@funinf.cs.unibuc.ro
Contribution of the paper  This paper focuses on the foundations of interactive systems, presenting rv-systems, a model for interactive systems based on register machines and space-time duality.

Register machines are a well-known useful model for both theoretical and practical classical computer science. A register machine may be seen as a combination of a finite automaton for control and registers for storing data. The aim of a research project, of which this paper is a key step, is to construct a similar model for interactive systems, using a combination of two automata for control and interaction, and registers and voices for data.

The formal language theory is an old, rich, and well-established field – see [11] and the other chapters of the Handbook; particularly, the models of 2-dimensional languages [6], including tiling systems, have strong connections to our model, providing a source of good results and proof techniques.

At an abstract level, a model for interactive systems, called finite interactive systems (FIS-es), was introduced in [14]. This level is of a medium complexity, e.g., emptiness problem is undecidable, but membership problem is decidable. The model is able to filter out some impossible scenarios present in the case simple finite automata are used for interaction and control, giving a finer approximation of the actual computations of an interactive systems.

Formally, FIS-es are equivalent with tiling systems (both specify the same class of grid languages). However, FIS-es come with the concepts of states and classes, which may be instantiated with registers and voices, being of a crucial importance for extending the model to the full class of interacting systems. Finite interactive systems are used as a backbone for describing the syntax and the semantics of our programming language for interactive systems.

A first advance made in this paper is to introduce a general model for interactive systems, called interactive systems with registers and voices (rv-systems). Briefly, rv-systems are obtained from FIS-es adding data for their spatial and temporal interfaces. The model is very powerful: it includes register machines, is space-time invariant, is compositional, may describe computations extending in both time and space, and is applicable to open, interactive systems. To achieve modularity in space the model uses voices (a voice is the time dual of a register) - they provide a high level organization of temporal data and are used to describe interaction interfaces of processes.

Next, we introduce a core programming language, built up on top of rv-systems. It consists of rv-programs (programs with registers and voices). We present the syntax and both an operational and a denotational semantics for this programming language.

Finally, the paper introduces a relational specification formalism for interactive systems based on high-level temporal structures on streams. As case studies, we present specifications for a simple OO-system and for a simple interactive game.

Related work  Rv-systems belong to a class of machine-oriented models of computation including: register machines, data-flow networks [16], asynchronous automata, etc. (Petri nets partially belong to this class.) Rv-systems inherit a key feature from Mealy-Moore machines, namely the “transducer view”. Rv-systems allow for “extension in space”, namely a potentially unbound number of processes may interact in a scenario row (macro-step). Notice that, in the above-mentioned models, the corresponding interaction is usually atomic and bounded by the transitions’ breadth. This unbounded interaction appear to be a key requirement for passing from concurrent to concurrent, OO systems; indeed, in the latter case, the chain of method invocations may be unbounded.
Actor calculi \[2\], classical process algebras, or more recent versions including \(\pi\)-calculus \[12\] and bigraphs \[9\], or tile logic \[5\] are more language-oriented - often, they are very powerful in describing process interaction. Usually, they are given an interleaving semantics described by transition systems or term models modulo some equivalences - testing equivalences \[4\] partially fit with rv-systems transducer view. Rv-systems may be seen as an attempt to close the language-machine gap by incorporating some of the actor and process algebra features into a machine-oriented model. Rv-systems have a true-concurrency semantics based on grids and scenarios. Due to their machine-oriented inheritance, rv-systems may be easier to implement on existing or slightly modified architectures.

UML is a popular and very powerful visual formalism. It provides many separate views for the system under construction, including statechart diagrams \[7\] and interaction diagrams. An integrated view of much simplified versions of statechart and interaction diagrams is captured by FIS-es, on which the syntax and the semantics of rv-programs is based. Rv-programs are simpler, so their theory may be more tractable. MSCs\(^1\) are often used for system specification in UML. Scenarios, seen as extensions of MSCs, may be used for programming too \[8\]. Compared with the above, our scenarios for rv-programs have a more relaxed view on time (see Section 4).

There are many specification formalisms used for timed systems - see, e.g., \[1, 3\]. Our high-level organization of temporal data based on voices, as well as their use in spatio-temporal specifications and rv-systems, look to be new. The time on our streams is finite, while in traditional models of reactive systems, the time is infinite; nevertheless, we hope some model checking methods apply to the current setting, too.

2. Grids and grid languages

2.1. Grids

**Grids** A grid \(^2\) is a rectangular two-dimensional area filled in with letters of a given alphabet \(V\). Their set is denoted by \(\mathcal{V}^{+}\).

Each letter in \(V\) is a two-dimensional atom having its own north, south, west, and east type. This typing is naturally extended to grids. More general grids may be used (removing the condition to have a rectangular area), but these general grids will be hardly used in this paper. It is important to notice that our grids are logical, not geometrical objects.

![Grids](image)

\(^1\)See the Appendix of \[15\] for a formalization of MSCs in the FIS-es setting.

\(^2\)In the literature, the term *picture* is often used as a substitute for grid or two-dimensional word, especially in the context of picture languages - see \[6\] and the web for pointers to papers on two-dimensional languages. However, our view is more semantical, hence we prefer to use the term ‘grid’ considered as a two-dimensional version of ‘path’. Sometimes we will also use the term ‘planar word’ (as a two-dimensional version of ‘word’), but never ‘picture’.
A few examples of grids are presented in Fig. 1. The grid in (a) is a normal, rectangular grid - by default, a grid is considered of this type; in (b), a more general grid is presented; finally, (c) describes our standard order used in grids: each cell directly depends on its top and left neighbors.

In our standard interpretation the columns correspond to processes, the top-to-bottom order describing their progress in time. The left-to-right order corresponds to process interaction in a nonblocking message passing discipline: a process sends a message on the right, then continues its execution. We will see later that the convention to send messages left-to-right only is not restrictive.

**Action vs. inter-action**  
The convention of having only a left-to-right causality in grids may raise an important question:

How *inter*-action may appear? A left process may send a message to a process on the right, but can not receive an answer back!

Or, put in other words, it looks that grids properly describe *actions* (like sending messages from masters to slaves), but not real *inter*-actions.

![Action vs. inter-action](image)

Let us recall that our grids are logical, not geometrical objects. A two-ways communication situation is isomorphic to the situation described in Fig. 2(a). If the alphabet of grid letters is rich enough to contain empty cell, identities, corners, and crossing, then, as in Fig. 2(b), the picture may be converted into a grid which faithfully captures the situation.

### 2.2. Scenarios

As we already said, grids are used to describe computations - a letter in a grid represents a statement to be executed. A *scenario* is a grid enriched with information about data used at the borders of its letters.

The additional information on data around each letter may be given in an abstract form as in this picture (a name *A, B, 1, 2*, representing a *type*)

1 1 1  
AaBbBbB  
2 1 1  
AaAaBbB  
2 2 1  
AaAcAaB  
2 2 2

or in a more detailed form as in Fig. 3.
We do not describe the details of a scenario like the one in Fig. 3 now. At this stage, just notice that the letters of the underlying grid are those in the boxes (X,U,V,…), while the neighboring areas are used to put extra information.

## 3. Finite interactive systems

### 3.1. Finite interactive systems

**Intuition** A finite interactive system may be seen as a kind of two-dimensional automaton mixing a state-transforming automaton with an automaton used for the interaction of the processes created by the first automaton. (Alternatively, one may consider a finite interactive system to be a kind of iterate transducer). Syntactically, it may be described by a graph as in Fig. 4.

While the above presentation may be somehow useful, it is also misleading. Actually, not two different automata are combined, but these two views are melted together to create the concept of finite interactive systems. For instance

- there are two FIS-es with the same projection automata, but different grid languages; and

- there are FIS-es with the same language, but non-equivalent projection automata.

(See [15] for examples.)
Finite interactive systems  A finite interactive system (FIS) is a finite hyper-graph with two types of vertices and one type of (hyper) edges:

- a first type of vertices is for states; we label them using numbers or lower case letters;
- the second type of vertices is for classes; we use capital letters as labels;
- the edges (also called transitions) are labeled by letters denoting atoms of the grids; they obey the following constraints: (1) each transition has two incoming arrows: one from a class vertex, the other from a state vertex, and (2) each transition has two outgoing arrows: one to a class vertex, the other to a state vertex.

Some classes/states may be initial (in the graphical representation this is specified by small incoming arrows) or final (the double circle representation is used).

One may use a semi-textual representation for FIS-es (e.g., $F_1$ in Fig. 4 is defined by: $A, 1$ initial; $B, 2$ final; and transitions $\begin{array}{c} 1 \\ A \end{array} \overset{a}{\rightarrow} \begin{array}{c} 2 \\ B \end{array}$) or a fully-textual one (e.g., specifying transitions as $(A, 1) \rightarrow (B, 2)$ or $a: (A, 1) \rightarrow (B, 2)$).

**Parsing procedure**  Given a FIS $F$ and a grid $w$, insert\(^3\) at the north (resp. west) border of $w$ initial states (resp. classes) and parse the grid selecting unprocessed atoms having a state (resp. class) on their north (resp. west) border. For each such atom $a$, if $s$ (resp. $C$) is its north state (resp. west class), then

---

\(^3\)At an abstract level, this kind of parsing is more or less conventional - one can build up a recognizing scenario starting with the south-east borders, too. Generally, there is no clear left-to-right top-to-bottom causality in abstract grids, as the example on spiral words suggests. However, this convention is essential for rv-programs as the programs are usually deterministic and the computation can not be reversed.
choose a transition $\begin{bmatrix} C \to \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·
4. Spatio-temporal specifications

4.1. Data with temporal representation

Turing tape was a first spatial memory model used in machine-oriented models of computation; complex data structures may be implemented on top of this simple model. We propose to use a similar approach to create complex data structures in time.

Voices

Registers holding numbers may be implemented on a Turing tape. At their higher level, the tedious aspects of identifying the positions of the numbers on the tape, the need to shift data when more space is needed, the computation of arithmetical or logical operations at the bit level, etc. are all hidden and one can get a more readable specification of the problem and of its solution.

Similarly, we start with a simple linear temporal data model - the stream structure. A stream, as used in the dataflow setting, is a finite or infinite sequence of data ordered in time. A stream is denoted as \(a_0 \prec a_1 \prec a_2 \prec \ldots\), where \(a_0, a_1, a_2, \ldots\) are its data (tokens) at time \(0, 1, 2, \ldots\), respectively.

The contents of our streams will be always finite, but unbounded in time, in the same way the contents of a Turing tape is always finite, but unbounded in space. One may think of a stream as the result of observing the data transmitted along a channel: it exhibits a datum (corresponding to the channel type) at each clock cycle.

A voice is a temporal structure that holds numbers.

Voices may be implemented on top of a stream in a similar way registers are implemented on top of a Turing tape. For instance, voices may be specified giving their starting addresses and their lengths: e.g., a voice \(v\) holding 2004 is defined by its temporal address \(t_0\) (its starting time on the stream) and its length 4 (if decimal representation is used) - this means, from that point in time \(t_0\), the cell corresponding to the stream is showing the digits 2,0,0,4 during 4 consecutive clock cycles.

At this new and higher level of abstraction we are interested in voices and their contents only. We are not interested in the implementation details as: representation (continuous time slots: 1st voice, 2nd voice, ...; or alternating digits: 1st digit of 1st voice, ..., 1st digit of last voice, 2nd digit of 1st voice, ...; etc.), position on the stream, low-level manipulation, etc.

More data with temporal representation

The above setting is good for theoretical purposes (a voice may represent an arbitrary long number), but in practice more concrete data structures are needed. Most of usual data structures have natural temporal representations; we add a “t” in front of normal types to denote these new temporal types. Examples: \(t\text{Bool}\) (booleans), \(t\text{Int}\) (integers, of various lengths), \(t\text{Array}\) (arrays), \(t\text{LinkedList}\) (linked lists), etc.

\(^4\)A Turing tape (or other memory organizations for standard data types used in imperative programming) exhibits a temporal dimension, too. By “spatial” we mean the memory has mainly a spatial extension and it is so organized that its cells may be reached in a constant amount of time. Then, a “temporal” data organization is a dual setting with a limited space, but large time extent.

\(^5\)The approach does not break so sharp with the tradition as it may appear at a first sight. High-level temporal data structure are present in conventional programs, even to a very large extent: they are represented by control structures, the very heart of programming. Indeed, if one picks up a statement and record the time when it is activated, then one gets a temporal datum. But this form of temporal data appear in an impure form, as control structures mix pure control with spatial data. To conclude, our proposal here is to strip control structures of their interaction with spatial data and to present temporal data in a pure form.
4.2. Specifications

Relational spatio-temporal specifications A few conventions: To distinguished between the products of data with spatial or temporal representation, we use the notation \( \cdot \otimes \), for the spatial product and \( \cdot \leftarrow \), for the temporal one (mathematically, they are just Cartesian product). Moreover, \( \mathbb{N} \otimes^k \) denotes \( \mathbb{N} \otimes \ldots \otimes \mathbb{N} \) (\( k \) terms) and \( \mathbb{N} \leftarrow^k \) denotes \( \mathbb{N} \leftarrow \ldots \leftarrow \mathbb{N} \) (\( k \) terms).

A spatio-temporal specification is a relation
\[
S \subseteq (\mathbb{N}^m_{\leftarrow} \times \mathbb{N}^p_{\otimes}) \times (\mathbb{N}^n_{\otimes} \times \mathbb{N}^q_{\leftarrow})
\]
between input and output registers and voices. It is denoted as \( S : (m, p) \rightarrow (n, q) \), where \( m \) (resp. \( p \)) is the number of input voices (resp. input registers) and \( n \) (resp. \( q \)) is the number of output voices (resp. output registers). On elements, it is defined as a relation between concrete tuples, written as \( \langle v \mid r \rangle \leftrightarrow \langle v' \mid r' \rangle \), where \( v, v' \) (resp. \( r, r' \)) are tuples of voices (resp. registers).

Examples The constants used in Fig. 2
\[
c0 = \varnothing, c1 = \Box, c2 = \Box, c3 = \Box, c4 = \Box, c5 = \Box
\]
have a natural relational interpretation:
\[
c0 = \varnothing;
c1 = \{ \langle \mid x \rangle \leftrightarrow \langle \mid x \rangle : x \in \mathbb{N} \};
c2 = \{ \langle x \mid \rangle \leftrightarrow \langle x \mid \rangle : x \in \mathbb{N} \};
c3 = \{ \langle \mid x \rangle \leftrightarrow \langle x \mid \rangle : x \in \mathbb{N} \} \text{ (space-to-time converter)};
c4 = \{ \langle x \mid \rangle \leftrightarrow \langle \mid x \rangle : x \in \mathbb{N} \} \text{ (time-to-space converter)};
c5 = \{ \langle x \mid y \rangle \leftrightarrow \langle x \mid y \rangle : x, y \in \mathbb{N} \}.
\]

Composing specifications Speciﬁcations may be composed horizontally and vertically, provided their types agree. For two speciﬁcations \( S_1 : (m_1, p_1) \rightarrow (n_1, q_1) \) and \( S_2 : (m_2, p_2) \rightarrow (n_2, q_2) \):

- the horizontal composition \( S_1 \triangleright S_2 \) is deﬁned only if \( n_1 = m_2 \); the type of \( S_1 \triangleright S_2 \) is \( (m_1 + p_2) \rightarrow (n_2, q_1 + q_2) \); the composite is deﬁned as expected:
  \[
  \langle v \mid r_1, r_2 \rangle \leftrightarrow \langle v'' \mid r'_1, r'_2 \rangle \text{ in } S_1 \triangleright S_2 \text{ iff }
  \exists v', \langle v' \mid r_1 \rangle \in S_1 \text{ and } \langle v'' \mid r_2 \rangle \in S_2 \text{ and } \langle v'' \mid r'_2 \rangle \text{ in } S_2
  \]

- the vertical composition \( S_1 : S_2 \) is deﬁned in a similar way - it is deﬁned only if \( q = p_2 \); its resulting type is \( (m_1 + m_2, p_1) \rightarrow (n_1 + n_2, q_2) \).

Let us make a comment on the type of composed speciﬁcations. It may look a bit strange to see that the type of composed speciﬁcation is expanded and one gets bigger and bigger types. However, this is natural when one tries to get a modular speciﬁcation and focuses on specifying parts of the system - the whole system may still have a reasonable small input-output type.

For instance, in an OO system one has to specify input data for all objects used in a run of a program - hence, speciﬁcations have these expanded types. Such a tedious task is avoided by: (1) using constructors providing initial data when objects are created and (2) collecting the ﬁnal results in the main program. In other words, except for the main program, the overall type of an object is \( (m, 0) \rightarrow (n, 0) \). Repeated horizontal composition of objects does not change the spatial type, which is reduced to the type of the main program.
4.3. Specifying a simple OO system

Object-orientation is a very rich field and it may be virtually impossible to touch even its most basic aspects in a brief subsection. The aim here is very modest: we show how a simple OO-system may be specified, how it may be decomposed, and discuss about the usefulness of temporal types.

Colored balls  The system under construction consists of colored balls (points) in a rectangular 2-dimensional area. A ball may be moved up-down and left-right using two buttons - one for the vertical, the other for the horizontal movement. The color of a ball is changed to “blue / yellow / red” when it touches the “north or south border / a corner / west or east border”. For simplicity, we use one ball and a $5 \times 5$ rectangle.

In the design of such a system we use: (1) three fields $x,y,z$ for $X$-coordinate, $Y$-coordinate, and color (for colors, use the code $9/0/1$ to represent “blue / yellow / red”, respectively); (2) two methods: $h(u)$ for horizontal movement ($u$ is $9/0/1$ for “move-one-cell-left / stay / move-one-cell-right”, respectively); $v(u)$ for vertical movement ($u$ is $9/0/1$ for “move-one-cell-down / stay / move-one-cell-up”, respectively); and (3) an attribute $c$ - it returns the color $z$.

Before working on the specification, we have to fix an interactive behavior of the system: the input interaction is via the horizontal-vertical movement buttons, while the output is just the display of the ball color.

Global specification  The specification uses

- a register for each field and
- a voice for each method or attribute (without loss of generality, we suppose a voice is able to represent the parameters of its associated method as a digit - just use a large enough set of digits$^6$; similarly for attributes).

The type of this global specification is $S : (3,3) \rightarrow (3,3)$, so we have to specify a set of tuples

$$\langle h, v, c \mid x, y, z \rangle \mapsto \langle h', v', c' \mid x', y', z' \rangle$$

An example is

$$\langle h_1, v_1, c_1 \mid 3, 3, 0 \rangle \mapsto \langle h_1', v_1', c_1' \mid 3, 3, 9 \rangle$$

where $h_1 = 11091990, v_1 = 90199911, c_1' = 01110999$ ($c_1, h_1', v_1'$ are irrelevant). Each position in $h$ and $v$ describes the status of the input buttons at a given clock cycle; they are independent, so any combination is possible; if $a^i$ denotes the $i$-th digit of a voice $a$, then at the $i$-th clock cycle the system passes from its state $(x,y,z)$ to a new state $(x',y',z')$ by actions $h_1^i, v_1^i$, written $x,y,z \xrightarrow{h_1^i,v_1^i} x',y',z'$; the system evolution is

$$3, 3, 0 \xrightarrow{19} 4, 2, 0 \xrightarrow{10} 5, 2, 1 \xrightarrow{01} 5, 3, 1 \xrightarrow{19} 4, 2, 1 \xrightarrow{19} 5, 1, 0 \xrightarrow{19} 4, 1, 9 \xrightarrow{91} 3, 2, 9 \xrightarrow{01} 3, 3, 9$$

This sequence gives the final state and it is also used to compute the voice $c'$ of the output colors.

$^6$If there are more parameters, then one may use a coding to represent a tuple of numbers by a unique number, then apply the above convention.
Horizontally (de)composed specification  The above specification may be horizontally decomposed and described in a modular way: Use a class for one-dimensional points in a bounded interval; these points may be moved in both directions, reporting when the border is touched; let hit be a voice used to record these events. Take two such points (one for horizontal, the other for vertical direction) and “compose” them to get the original system; the link is realized using an additional object which receives the hit information to generate the final color. Formally, take

\[ S_1 = S_h \triangleright S_v \triangleright S_c \]

where

- \( S_h : (5, 1) \rightarrow (5, 1) \) is defined by: \( \langle h, v, c, hith, hitv \mid x \rangle \mapsto \langle h, v, c, hith', hitv \mid x' \rangle \)
- \( S_v : (5, 1) \rightarrow (5, 1) \) is defined by: \( \langle h, v, c, hith, hitv \mid y \rangle \mapsto \langle h, v, c, hith, hitv' \mid y' \rangle \)
- \( S_c : (5, 1) \rightarrow (5, 1) \) is defined by: \( \langle h, v, c, hith, hitv \mid z \rangle \mapsto \langle h, v', c, hith, hitv \mid z' \rangle \)

for appropriate mapping describing the primed variables (i.e., hith', x', . . .) in terms of the inputs. \( S \) is obtained restricting \( S_1 \) to its first three voices \( h, v, c \).

To have an example, case (1) above is decomposed as follows (0/1 is the code for non-hit/hit event):

- with \( h_1 \) in \( S_h \), one gets hith'\( _1 = 01101000 \) (and \( x'_1 = 3 \))
- with \( v_1 \) in \( S_v \), one gets hitv'\( _1 = 00001100 \) (and \( y'_1 = 3 \))
- with hith'\( _1 \) and hitv'\( _1 \) in \( S_c \), one gets \( c' = 01110999 \) (and \( z'_1 = 9 \))

Usefulness of temporal types  The above specification is based on a global typing for the temporal data, i.e., all components \( S_h, S_v, S_c \) are allowed to see (or “hear”) all voices. This peculiarity may be used to describe specifications where the components cheat in their interaction.

In the above decomposition \( S_h \) may cheat: while not his business, he may change the content of \( v \), preventing component \( S_v \) of having a right reaction. For instance, using (the assumption that initially \( y = 3 \) and) \( \langle h, v, c, hith, hitv \mid x \rangle \mapsto \langle h, v', c, hith', hitv' \mid x' \rangle \) with \( v' = 91919191 \) for any input \( v \), \( S_h \) fakes the meaning of the vertical button, preventing \( S_v \) of touching the borders - now, \( S_v \) keeps moving back and forth.

A simple solution to avoid such abnormal behavior is to use more restricted temporal types. From the analysis of the system, we may provide a more restricted typing. The initial specification has the restricted type \( S : (2, 3) \rightarrow (1, 3) \) mapping \( \langle h, v \mid x, y, z \rangle \mapsto \langle c \mid x', y', z' \rangle \). It may be decomposed as

\[ S = (S_h \sim Id) \triangleright (Id^{-} S_v) \triangleright S_c \]

where \( Id \) denotes appropriate identities and \( S_h, S_v, S_c \) are (with appropriate mappings for hith, x', . . .)

\[
\begin{align*}
S_h : & \ (1, 1) \rightarrow (1, 1), \text{ defined by } \langle h \mid x \rangle \mapsto \langle hith \mid x' \rangle \\
S_v : & \ (1, 1) \rightarrow (1, 1), \text{ defined by } \langle v \mid y \rangle \mapsto \langle hitv \mid y' \rangle \\
S_c : & \ (2, 1) \rightarrow (1, 1), \text{ defined by } \langle hith, hitv \mid z \rangle \mapsto \langle c' \mid z' \rangle
\end{align*}
\]

This kind of horizontal decomposition looks to be related to OO class manipulation, but the precise relationship still has to be investigated. It is a very powerful mechanism and it is supported in rv-programs by a kind of “programming in space” paradigm.
Notice that $S_h \sim Id : \langle h, v \mid x \rangle \mapsto \langle h, v \mid x' \rangle$ and $Id \sim S_v : \langle h, v \mid y \rangle \mapsto \langle h, hv \mid y' \rangle$, so the type of the right-hand side is $\langle h, v \mid x, y, z \rangle \mapsto \langle c \mid x', y', z' \rangle$, equal to the type of $S$.

With such a detailed decomposition and restricted typing mechanism, a cheating as above is no more possible.

This OO-system specification is flat, i.e., only one level of the interaction is considered. If we look to a scenario (for FIS-es or rv-programs), we see that during their lifetime two processes (columns) may interact via different classes, each with its particular temporal type. Let us emphasize this interesting observation:

**Fact 4.1.** Temporal typing is changing in time; a concrete temporal type for process interaction lasts for a limited time period.

### 4.4. Specifying an interactive game

This subsection describes a spatio-temporal specification of a simple interactive game. An additional aim of the analysis is to get a better understanding of the temporal aspects of these specifications looking to a “local vs. global time” dilemma.

**01-game** The game uses an $n \times n$ table and 2 players, one using ‘1’, the other ‘0’. The players alternate placing their symbols on the table (one symbol at a time). To win the game, a player has to place 3 consecutive symbols on the same line, column, or diagonal. To keep the specification simple, we restrict ourself to the trivial case $n = 3$. The cells are identified by numbers 1 to 9 (counting: left-to-right, then top-to-bottom).

**The specification** The specification $S : (2, 1) \rightarrow (2, 1)$ is defined by $\langle p0, p1 \mid t \rangle \mapsto \langle a0, a1 \mid t \rangle$ where

- $t$ is a register to record the status of the table, i.e., the $i$-th digit of $t$ describes the content of the $i$-th cell. The code is: 9 for empty cell, 0 for cell holding 0, and 1 for cell holding 1.
- $p0, p1$ are input voices to record players’ actions and $a0, a1$ are output voices to score the results.

In $p0, p1$ the code is: 0 for no action and $k \in \{1, \ldots, 9\}$ for placing the symbol on the $k$-th cell. In $a0, a1$ the code is: 9 for wrong-move (or loss), 0 for good move, and 1 for win.

A few examples are presented below (star means ‘don’t care’): (a) describes a complete play ending with a tie; (b) describes a better strategy for player 0, which wins; in (c), player 1 does not provide a move when it is his turn, so player 0 wins; (d) describes a case where both players move at the same time and player 2 loses (it was not his turn).

(a) $\langle 503040801, 060207090 \mid 999999999 \rangle \mapsto \langle 000000000, 000000000 \mid 010001101 \rangle$
(b) $\langle 50208****, 06010**** \mid 999999999 \rangle \mapsto \langle 00001****, 00009**** \mid 10**01*0* \rangle$
(c) $\langle 5030****, 06000**** \mid 999999999 \rangle \mapsto \langle 0001****, 0009**** \mid ***0*01**** \rangle$
(d) $\langle 5030****, 061**** \mid 999999999 \rangle \mapsto \langle 001****, 009**** \mid ***0*01**** \rangle$

If one sticks to have a functional specification, than star should be avoided and a more detailed and restricted functional specification has to be given.
We included a few generic cases to illustrate the method. It is clear that a full specification of the game may be described using this formalism.

The specification language looks to be quite powerful: one may specify a situation where both players move at the same time, or one player misses his move, etc. So, let us concentrate on a few aspects related to possible lower level implementations of the specifications (schedule) to get more insights on these high-level specifications.

A lesson from register machines A spatio-temporal specification describes a relational transformation between the input registers and voices and their output counterparts. Voices are specified in time and, at a lower level, we have to schedule them on streams. For our 01-game one may find an arrangement of their actions in time to fit the intuition: e.g., the configuration in (a) above (503040801060207090) may be represented by \( 5_{6}^{0}0_{2}^{4}0_{0}^{7}8_{0}^{9}0_{9}^{0}1_{0}^{0} \).

However, notice that the input voices will be scheduled on the input stream, while the output ones on the output stream. In a real situation, it is supposed that the players will see the results before giving a next move. So, it should be a relation between the time used on the input stream and the time used on the output stream.

Let us recall the situation with register machines. In that case, the custom is to clearly differentiate between input and output registers, for instance using different variables. As they are different, when we implement them on a tape there is no constraint to have a specified relation between their positions on the tape.

To have modular high-level spatio-temporal specifications, perhaps we have to follow a similar rule by not relating the physical time on the output stream to the physical time on the input stream.

The blind player The above argument may not be fully convincing. However, there is a stronger argument leading to a similar conclusion - see below.

As we said, in real situations it is supposed that the players will see the results before giving a next move. Now, looking more carefully to the statement, we see that we actually introduce assumptions about the players, namely regarding their sight capabilities. However, it is not a good idea to mix the relatively easy task of specifying a simple game with the daunting task of specifying the capabilities of a human player!

Here we may apply a reasoning somehow similar to that used in Turing test\(^9\): Suppose we have a blind player which “by pure chance” choose to play the same moves as a player with sound sight capabilities. From the point of view of the game, only the moves matter, so there is no difference between having a blind or a sound player, as long as they make the same moves. Once we have reached this point, the next step of the argument is easy to guess: for the blind player, there is really no difference on whether the output of a move is shown before the next move or not.

Relaxing time constraints To conclude, in our spatio-temporal specifications it is desirable to weaken the assumption of having a clear and tight relationship between the physical time on the input stream and the physical time on the output stream. Then,

---

\(^9\)Turing test on natural vs. artificial intelligence.
Proposition 4.1. With a local time view, if enough memory space is provided, then any computable relation between input and output voices may be implemented.

Proof:
Indeed, if enough memory space is provided, then one may record the contents of all voices, then do the transformation within a conventional state-transforming model, and finally deliver the results in time.

The assumption of having enough memory may be practically violated. Then, at a lower implementation level, we have to refine the specification and to come up with a detailed program where the physical timing is essential. But this refinement will come later: it is not part of the starting abstract specification, it will be the very heart part of the implementation.

5. Rv-systems and rv-programs

Definition 5.1. (rv-system)
An rv-system (interactive system with registers and voices) is a FIS enriched with:

1. registers associated to its states and voices associated to its classes; and

2. appropriate spatio-temporal transformations for letters.

A computation is described by a scenario like in a FIS, but with concrete data around each letter.

In the following we focus on the class of programmable rv-systems, more precisely those described by rv-programs.

5.1. Rv-programs

Rv-programs - a minimal instruction set
Register programs can compute all partial computable functions using a small set of instructions, e.g. plus-1 (add 1 to a register), minus-1 (subtract 1 from a register), and test-0 (test if a register is 0); moreover, the last two instructions may be combined, reducing the number of instruction types to 2. One may introduce rv-programs using a similar small number of instructions.

Definition 5.2. (rv-program, theoretical setting)
An rv-program (program for interactive systems with registers and voices) uses a finite set of registers and voices. It consists of a list of statements of the following type

\[(A,a): x = x+1 \text{ goto } [B,b]; \text{ or} \]
\[(A,a): \text{ if } x = 0 \text{ goto } [B,b]
\[\text{ else } x = x-1 \text{ goto } [C,c];\]

where \(x\) is a register or a voice and \(A,B,C,a,b,c\) are labels.

We skip the details, examples, etc. as this is part of a much expanded and more practical setting described below.
Rv-programs - a more practical setting  The running example is more or less random: as factorial function is a popular example used to illustrate various programming concepts, we are using here a kind of “interactive factorial function”.

The program, denoted MAFact, starts with an input \( n \) (of type \( \texttt{sInt} \) - spatial integer) and computes

\[
f(n) = ((1 \ast n) \ast (n - 1)) \ast \cdots \ast (n_{-1}) 2 \ast n 1
\]

The operation to be performed at a step \( i \) is either multiplication or addition and it is known by reading the values of the interaction variables \( m, a \) (of type \( \texttt{tBool} \) - temporal booleans) at stage \( i \): if \( m \) is true, then multiplication is chosen, otherwise addition (provided \( a \) is true). Moreover, multiplication is implemented using additions (to illustrate how computation may be extended in space). Its actual code is shown in Fig. 5.

Syntax

Definition 5.3. (rv-program, practical setting)  The syntax is based on the syntax used in imperative programming languages. The basic block is a module. To explain the syntax, let us focus on the first module of MAFact. It has a name \( X \) and 4 areas.

1. In the top-left part we have a pair of labels \( (A,1) \) which specifies the interaction and control coordinates where this module has to be applied. Notice that similar pairs of labels are used inside the code (described in the bottom-right area), but the format \( [B,3] \) and the meaning are different. We will return to this later.
(2) The top-right part declares the spatial input variables. These variables specify the memory state before the application of the module. To distinguish them from the variables used for interaction interfaces, we put an “s” in front of their types. For module X there are two spatial input variables of type integer, denoted sInt.

(3) The bottom-left part is similar to the top-right one, but it declares the temporal input variables. These variables are used for the data appearing to the interaction interfaces between modules. We put a “t” in front of normal types to indicate these data have temporal representations. Module X has two temporal input variables of type boolean, denoted tBool.

(4) The body of a module is its bottom-right part. It may include type declarations for new variables and the code to be executed by the module, similar to C code. The computation within this part makes no distinction between a spatial and a temporal variable. The exit from the module is realized by a goto statement. The computation within an interactive systems has two dimensions - one vertical, the other horizontal. A statement like goto [B, 3] indicates that:

- The data of the spatial variables in the current module will be used in a next module with control state 3. (What will be the new interaction data is not known from this module alone - it depends on the global properties of the interactive system.)
- The data of the temporal variables in the current module will be used for the interaction interface of a new module with interaction label B. (Again, it is not known from this module alone what will be the control state of the new process when it receives these data at its temporal interface.)

5.2. Semantics

Operational semantics The operational semantics of rv-systems is given in terms of scenarios. For MAFact, a first example of such a scenario is illustrated in Fig. 3 - it corresponds to a particular case where the (spatial) input variable n is 4 and the interaction interface A keeps choosing m to be true, hence the result is (((1 × 4) × 3) × 2) × 1 = 24. A second scenario is illustrated in Fig. 6 - this latter case is like the former, but in the 3rd round m is false and a is true, hence the result is (((1 × 4) × 3) + 2) × 1 = 14.

Definition 5.4. (scenarios)
Scenarios are constructed using the following procedure (see, e.g., the scenario in Fig. 6):

1. Each cell of the corresponding grid has a label in \{X, Y, Z, U, V, W\} specifying the module of the program used in that particular cell. A cell has associated states on its top and bottom neighboring areas and associated classes on its left and right neighboring areas. The grid is constructed starting with its top row and left column (containing the inputs) and progressively inserting new cells when their left and top neighboring areas are already computed/updated.

2. An area may have an additional information as s=2 - that means, in that area s has been updated to be 2.

3. The full information on the current state of a process is obtained going vertically up and collecting the last updated values of the spatial variables. For instance, the bottom Z in the first column has n=3, y=1.
Figure 6. 2nd scenario for \texttt{MAFact}

(4) The full information on temporal variables in a current place is obtained collecting their last updated values going horizontally on left. For instance, \(X\) in the 6-th column has \(m=1, a=0\).

(5) The 1st column has an input class (here \(A\)) and particular tuples of values for its temporal variables. The 1st row has an input state (here \(1\)) and particular tuples of values for its spatial variables.

(6) Once the above points were clarified, the local computation of a cell \(\alpha\) is easy to describe:

(a) Take a module \(\beta\) of the program bearing the class label of the left neighboring area of \(\alpha\) and the state label of the top neighboring area of \(\alpha\).

(b) Follow the code in \(\beta\) using the spatial and temporal variables of \(\alpha\) with their current values.

(c) If the local execution of \(\beta\) is finished with a \texttt{goto} \([\Gamma, \gamma]\) statement, then the label of the right neighboring area of \(\alpha\) is set to \(\Gamma\) and the label of the bottom neighboring area of \(\alpha\) is set to \(\gamma\).

(d) Insert in the right neighboring area of \(\alpha\) the values of the temporal variables updated by \(\beta\) and in the bottom neighboring area of \(\alpha\) the values of the spatial variables updated by \(\beta\).

(7) A \textit{partial scenario} (for an rv-program) is a scenario constructed using the above rules; it is a \textit{complete scenario} if the bottom row has only final states and the rightmost column has only final classes.

Both scenarios in Fig. 3 and Fig. 6 are complete scenarios for \texttt{MAFact} program.

The \textit{operational semantics} of rv-programs is defined in terms of scenarios: for a given program and concrete data for its registers and voices, the behavior is specified by the corresponding scenario. Notice that in this semantics we used a denotational semantics for the body of the modules. If one stick to have
New in-out type: in: E,5,n,mt[n],at[n]; out: F,C,4,y

Change in module X: add write y before goto [C,4];

Add new modules:

X1::
(F,5)

<table>
<thead>
<tr>
<th>(E,5)</th>
<th>n: sInt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mt[n], i: sInt, ms[n], as[n]: sArray of sBool;</td>
</tr>
<tr>
<td></td>
<td>at[n]: read n;</td>
</tr>
<tr>
<td>tArray</td>
<td>listen mt[1..n], at[1..n];</td>
</tr>
<tr>
<td>of</td>
<td>ms[1..n] = mt[1..n]; as[1..n] = at[1..n];</td>
</tr>
<tr>
<td>tBool</td>
<td>i = 1; goto [F,6];</td>
</tr>
</tbody>
</table>

X2::
(F,6)

<table>
<thead>
<tr>
<th>(E,6)</th>
<th>n,i: sInt, ms[n], as[n]: sArray of sBool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>m,a: tBool;</td>
</tr>
<tr>
<td></td>
<td>m = ms[i]; a = as[i]; i = i+1;</td>
</tr>
<tr>
<td></td>
<td>if (i &gt; n) then goto [A,4] else goto [A,6];</td>
</tr>
</tbody>
</table>

X3::
(F,5)

| (F,5) | p: tInt; |
|-------| n: sInt; |
| tInt | n = p; goto [F,1]; |

Figure 7. MAFact with input-output

a fully operational semantics in terms of more elementary operations, then one has the include execution paths for the code of the modules, too.

Input-output denotation The operational semantics may be used to define input-output denotation of rv-programs. Given its restricted syntax\(^{10}\), each module has a clear input-output denotation, i.e., it specifies a relation \( R \subseteq (\mathbb{N}^\sim \times \mathbb{N}^\sim p) \times (\mathbb{N}^\sim n \times \mathbb{N}^\sim q) \) between input registers and voices and their output counterparts. These denotations are lifted to the scenario level using the composition operators on relations defined in Sec. 4.2. The input-output denotation of an rv-program is the result produced by the above procedure.

The type of these denotations is not unique; e.g., in MAFact the temporal interface needs more or less data depending on the value of \( n \). Actually, MAFact is more like an internal piece of code, rather than an independent program with a clear input-output interface.

Input-output operations are specified by read/write instructions for spatial data and listen/speak instructions for temporal data.

The changes needed for MAFact to accommodate input-output instructions are described in Fig. 7. To shrink the temporal interface we collect all booleans \( m, a \) in boolean arrays\(^{11}\) and listen them at the

---

\(^{10}\)As we said, within the body of a module there is no difference between a spatial and a temporal variable. Consequently, the module body is a piece of code in a traditional imperative programming language and one may use classical techniques to find its input-output relation.

\(^{11}\)For notational convenience, the range of indexes for these arrays is denoted 1 .. n, not 0 .. n-1 as in C.
top-left cell of scenarios. Then, internally the program supplies the temporal data needed by the previous
version of MAFact program at its A interface.

The input output transformation corresponding to the modified MAFact program is
\[
\langle m[1..n], a[1..n] \mid n \rangle \mapsto \langle \mid ((\ldots((1 \ast_1 n) \ast_2 (n - 1)) \ast_3 \ldots \ast_{n-1} 2) \ast_n 1) \rangle
\]
where \( \ast_i \) is ‘\( \times \)’ if \( m(i) = 1 \) and ‘\( + \)’ if \( m(i) = 0 \land a(i) = 1 \).

5.3. Space-to-time/time-to-space converters

The information may be converted from a spatial to a temporal representation and vice-versa. An ex-
ample is present in Fig. 6: the value of the spatial variable \( n \) in the 1st column is copied in a temporal
variable \( p \), communicated to the 6th column, and converted back to the spatial variable \( n \). This is an
instance of a more general procedure where: one may copy the memory state of a process (or a part of
it), communicate it using temporal variables to a different process, and reuse there to (re)set the memory
state.

Let us return to Fig. 2 for a deeper understanding of modeling a two-ways communication. We
described the relational semantics of the involved constants (identities, corners, etc) in Sec. 4.2. The full
state of Process \( a \) in the 1st column is transmitted in time to the 4th column, then \( a \) terminates. In the
4th column the information is received in time and Process \( a \) is recreated. Normally, as process creation
and termination are expensive, a compiler will detect and avoid this termination-recreation situation by
reusing Process \( a \). There are subtle differences when the system is distributed and it happens the 1st and
the 4th columns in Fig. 2 describe processes located on different machines. Then the termination and
recreation are effective and Process \( a \) migrates from one machine to another.

6. Space-Time Duality

Space-time duality interchanges information in space and information in time, e.g., registers and voices.
Then, it is naturally lifted to grids, scenarios, FIS-es, spatio-temporal specifications, rv-systems, and
rv-programs which are all space-time invariant.

Definition 6.1. (space-time duality on rv-programs)
We define a space-time operator \( \checkmark \) by:

- On grids: transpose the grid; replace each letter by a dual letter;

- On FIS-es: interchange states and classes; replace each letter by a dual letter;

- On scenarios: apply \( \checkmark \) to the underlaying grid; around each letter interchange input registers with
input voices and output registers with output voices;

- On rv-programs, in each module: interchange class and state labels; interchange temporal and
spatial data; switch top-right and bottom-left corners; (notice that, except for label and variable
type change, no more modifications are needed in the body of a module).

\(^{12}\)Actually, this is the intended behavior of the program; the program is slightly ‘incorrect’, i.e., it does not terminate when the
last chosen operation is addition.
The proof of the following result is easy, as the model of rv-systems was build up using the space-time duality principle; due to its important consequences, we label it as a “theorem”.

**Theorem 6.1.** For any rv-program $R$, its space-time dual $R^\vee$ is an rv-program and $(R^\vee)^\vee = R$. Moreover, space-time respects operational semantics and input-output denotation.

This result suggests to introduce a new programming language construct: stdual. Applied to a piece of code, stdual interchanges space and time according to the above rules. For instance, if one has an estimation of the time spent by a usual piece of code $P$ and it is too high, than using

\[
\text{if (time}(P) < \text{tooHigh) then P else s2t(memory}(P)); \text{stdual}(P); \text{t2s(memory}(P))
\]

it may replace the computation in time of $P$ with a computation in space of $P^\vee$. (Here, s2t and t2s are notation for “space-to-time” and “time-to-space” converters).

This is a “heavy” and probably difficult to implement programming construct.

7. **Future work**

There is a large number of research directions and open questions, which may be classified into two main groups. On the one hand, there is a huge amount of practical and theoretical work dealing with interactive systems. As our model is new, its strengths and limitations are still to be investigated and compared with those of related models. On the other hand, there are intrinsic problems generated by this approach. A sample list from the latter group is included below.

**Exploit space-time duality:** Lift concepts and results from classical programming to interactive systems, sticking on getting space-time invariant models, which is a key feature of our approach here. Examples: lift structured programming (use it for rv-programming in space and clarify the relationship to OO class manipulation); develop assembly languages (low-level versions of rv-programs at bit level), appropriate architectures (register-and-voice based architectures), and compiling techniques.

**Representing FIS languages:** Find a Kleene theorem for FIS languages (use expressions without homomorphisms), study its associated algebras (keeping an eye on Jones’s planar algebras [10]), and use it to develop verification techniques for rv-programs.

**Typing systems:** Develop typing systems for rv-programs, focusing on types for temporal data and their role to security protocols.

**Develop programming languages:** Many challenges, including: how to write programs avoiding labels; how to integrate OO-programming; develop interpreters/compilers; select a focus area for applications (possible candidates are: cluster/peer-to-peer/grid computing, agents and e-commerce, interactive games, natural language processing, or biological processes).

**Acknowledgments** The author acknowledges with thanks the excellent working condition at the School of Computing, National University of Singapore, where part of these results were obtained and a first draft [15] has been completed.
References